United States Naval Observatory

DRAFT

Science Operations Center System Software Development Plan

DRAFT

7 September 2001

USNO-FAME-SOC-SDP
Version 0.2

[Signature page placeholder]

[Table of Contents placeholder]

1. Scope

1.1. Identification

This Software Development Plan (SDP) addresses the strategies and tactics for bringing about the creation of the Full-sky Astrometric Mapping Explorer (FAME) Science Operations Center (SOC) software.

1.2. System Overview

The purpose of the SOC is to

1. accept data from the NRL Ground Station at Blossom Point, MD,
2. provide real-time visibility into the FAME instrument health and status,
3. archive the raw data stream,
4. produce science data products including a full-sky star catalog.

Figure 1 shows the conceptual diagram for the SOC.

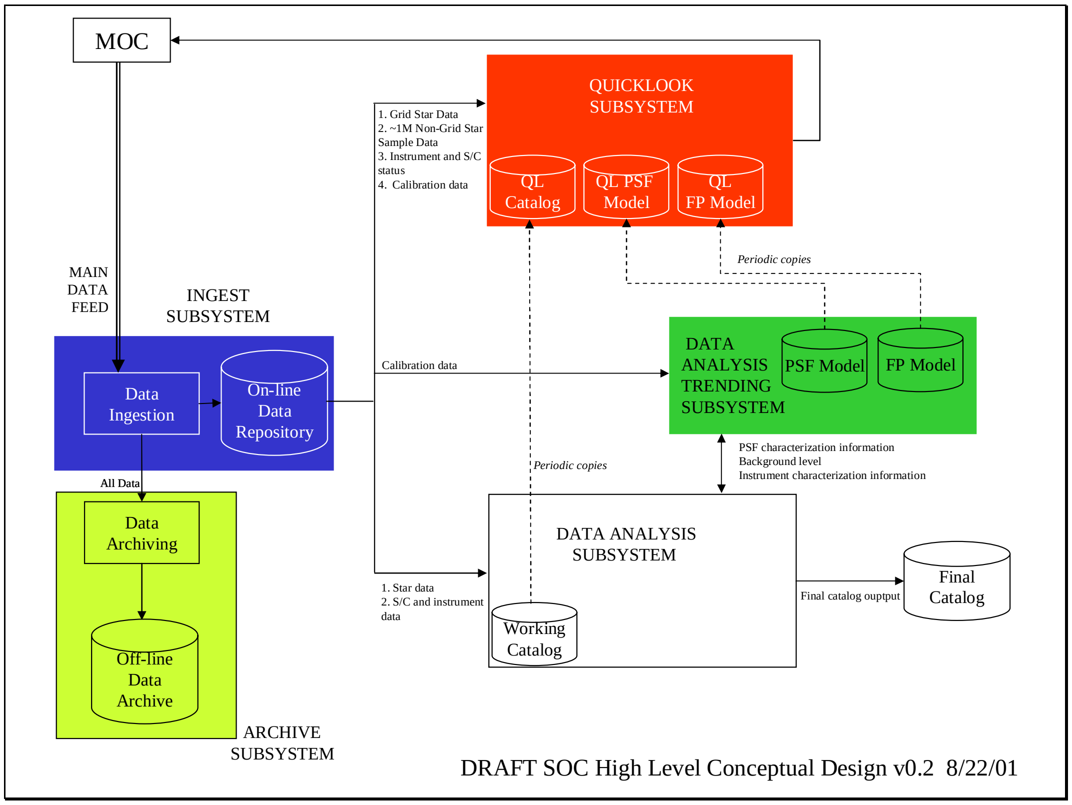


Figure 1. SOC Conceptual Design

The SOC consists of five subsystems:

1. **Ingest**
   Provides a means whereby the incoming data stream (telemetry) is routed to the appropriate subsystems.

2. **Quick Look**
   Performs real-time processing of data stream to provide quick visibility into the science data.
3. **Instrument Trending**
   Provides visibility into the optical focus, focal plane state, etc.
4. **Data Archive**
   Provides secure, long-term storage of the original data stream.
5. **Astrometric / Photometric Data Analysis**
   Handles formation of the high-resolution astrometric and photometric science products including the full-sky star catalog.

1.3. Document Overview

This document follows the Data Item Description (DID) for Software Development Plans as described in Mil-Std-498. It deviates from that guideline only in areas where the DID addresses topics not applicable to the FAME SOC. It is divided into the following parts:

1. Introduction (Sections 1 – 3)
2. General and Detailed Plans (Sections 4, 5)
3. Administration (Sections 6, 7)

1.4. Relationship to Other Plans

This document is a follow-on to the FAME Software Management Plan, v2, 13 Jun 2000, produced by the FAME Software Management Working Group. It capitalizes on work done in preparation of that document and expands the concepts to include technology advances and changes in the operations concept.

2. Referenced Documents

This SDP references these FAME preliminary design documents.

Document	Version	Date
SOC System Operations Concept	v0.2	28 Aug 2001
SOC System Requirements	v0.1	TBD
SOC System Preliminary Design	TBD	TBD
FAME Software Management Plan	V2	13 June 2000

3. Overview

The FAME SOC will satisfy the requirements by using a highly modularized architecture. Modularized code enforces data privacy (no public data) and communication only through specific module interfaces. This will allow much easier modification of the various parts without affecting other subsystems.

Figure 2 shows the general layout of the FAME SOC System from an implementation point of view.

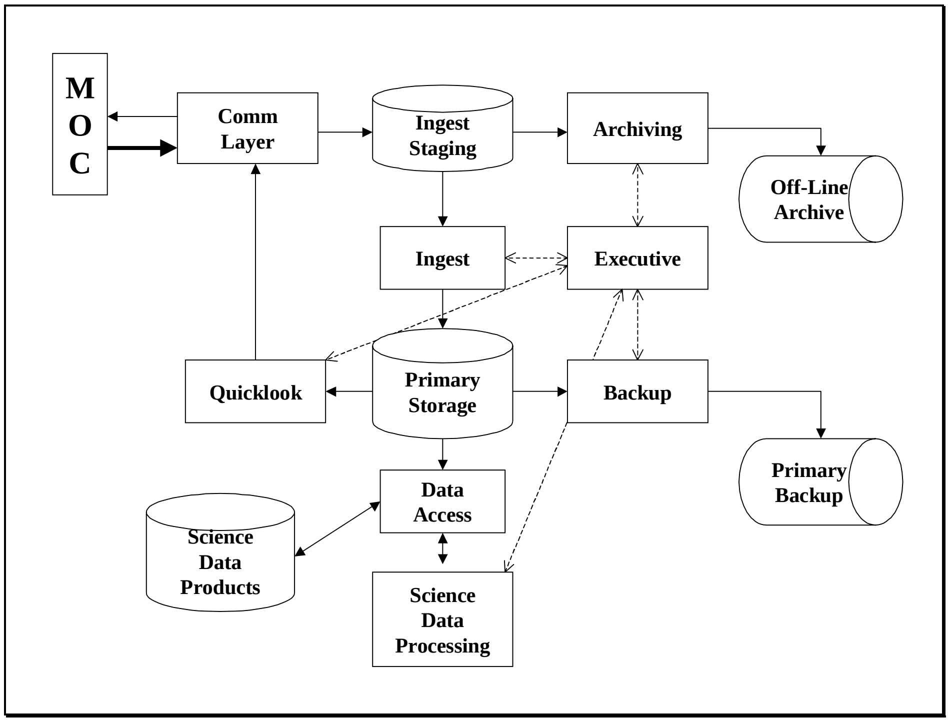


Figure 2. SOC High Level Architecture

The primary difference between the conceptual design and the High Level Architecture is the addition of the Communications Layer and the Data Access layer. The Communications Layer is added in order to centralize all communications with the MOC. This way, if the communication ICD changes in some way, the actual processing is insulated from the effect. Additionally, while the Ingest and Quicklook processes will be tightly coupled with the Primary Storage, by putting a Data Access layer between the Science Data Processing and the actual data, we allow the actual data storage system to change without impacting the Science Data Processing code.

4. General Plans

4.1. Process Overview

The FAME Software Development Process is derived from industry standard object-oriented modeling, design and development guidelines. FAME will use the Rational Rose product for system modeling and design and the TBD programming language for development.

The process consists of five phases:

1. System Analysis and Modeling
   Analyze the user community's ideas for what is needed from the system (user stories) in order to produce use cases and system requirements. Use these requirements to develop the models for system conceptualization, technical

architecture and preliminary design. This phase ends at Preliminary Design Review (PDR).

2. Detailed Design
   Expand the preliminary design by providing detailed descriptions of the individual modules to be produced. This design must fully specify the software to the level where individual software developers can complete the coding of the system. This phase will include the development of test procedures, test data and expected results to be used in the next phase. This phase ends at Critical Design Review (CDR).
3. Development
   This is the actual coding and integration of the system. During this phase, off-the-shelf systems are integrated with custom coded modules. This phase ends with the delivery of a working version of a fully functioning system for testing.
4. System Test and Delivery
   Test the completed systems against the final set of requirements. Review and revise as needed in order to satisfy all system requirements for that version.
5. Production and Maintenance
   Transition the system into the production environment and process the live instrument data. Analyze, accumulate and address problems that arise during production. Due to the nature of discovery experiments such as this, we expect a high number of post-launch modifications. These modifications are expected to have the highest impact on the non-launch-critical facets of the system.

4.2. Development Plans

4.2.1. Development Methods

Because of the skill mix present in the FAME development team, there will be several development methods in play. Using an object-oriented approach allows the use of state-of-the-art design and development tools and methods, however the learning curve associated with it is prohibitive on the projected FAME timeline. During prototype development, exploration will be done using a variety of languages, including C, C++, Fortran and IDL. The target language for production development will be C and C++. While a single language is preferable, the choice of these two allows for the varying levels of expertise in the FAME development community.

4.2.2. Standards

4.2.2.1.Coding Standards

The FAME project will use coding standards as defined in the FAME Software Coding Standards described in Appendix A. These standards were adapted from the original FAME Software Management Plan. The standards are written specifically for C++ coding. We have chosen this standard since it can be used with both C and C++ code. The prototypes will be developed in several languages, however we anticipate the final version of the code to be in both C and C++.

4.2.2.2.Data Format Standards

The main data coming from the MOC (and that data synthesized by the simulator) will conform to the FITS data format standards. The initial data model is the one developed by USNO Flagstaff and will be revised as needed. Data formats internal to the SOC are TBD.

4.2.3. Software Reuse

4.2.3.1.System Modularity

The SOC system components will be highly modular. This means there will be well-defined interfaces between the modules, a high level of data hiding (insulation of one module's data from access by other modules) and the universal use of "get" and "set" operations for data access. This means Module B will only be able to access Module A’s data by asking for it through a “get” operation; likewise Module A’s data may only be changed through use of a “set” operation. In this way, each module has absolute control over its own data.

4.2.3.2.Data Layer Interface

Because of the large volume of data being generated by FAME as well as the desire to have instant access to all members of the dataset we will be evaluating several mass data storage technologies. In order to eliminate the effect of trying various technologies on the bulk of the processing software, we will isolate the science data processing software from the actual data store by creating a data layer interface. Only this data layer will access the actual data store. This data layer will have a constant interface specification which will allow the developers to design to a fixed standard rather than having to recode or redesign every time the physical data store changes.

4.2.3.3.Planning for Software Reuse

While there are no specific requirements for software reuse, the documentation and modularity standards in place on the project will allow for a high potential for use on future projects.

4.2.4. Handling of Launch Critical Requirements

4.2.4.1.Security

The security requirements for the FAME SOC can be found in sections TBD of the SOC Systems Requirements Document.

4.2.4.2.Data Integrity

The data integrity requirements for the FAME SOC can be found in sections 3.2.2, 3.11.2, and TBD of the SOC Systems Requirements Document. In general, the FAME SOC is required to maintain an archive of the data as received by the SOC as well as archives of science data products as they become available. The SOC is also required to check the data for consistency and fidelity to ensure data is not being corrupted at some stage of the process.

4.2.4.3.Ingest Stability

The ingest stability requirements for the FAME SOC can be found in sections 3.2.1, and TBD of the SOC Systems Requirements Document. In general, the FAME SOC will provide a robust data ingestion process which will consistently accept new data and ensure that data is not corrupted or lost.

4.2.4.4.Data Quick-Look

The system shall provide a facility for real-time processing of the full dataset. This will allow the FAME science team to check the instrument for various status and health metrics. See the FAME SOC System Requirements Document for detail.

Quick Look will be prototyped before PDR so that we will have a high degree of confidence in our ability to satisfy these requirements.

4.2.5. Hardware Resource Utilization

All hardware purchased for and used by the FAME SOC will be used exclusively by the FAME SOC for FAME data processing and archival.

5. Detailed Software Development Plans

5.1. Project Planning and Oversight

5.1.1. Software Development Planning

FAME SOC software development will follow a hybrid structured / object oriented methodology. This hybrid methodology is the result of the desire to use industry best-practices and the constraints of manpower and skill mix. Object oriented methodologies are best suited for complex systems that may require modification as it progresses. Structured methodology is more intuitive and more readily understood by members of the FAME development team. Planning for the development process will consist of the following steps and milestones:

Pre-PDR:

1. Study the user stories and science requirements to determine the list of system level requirements,
2. Determine the rough scope of the development effort,
3. Evaluate options for system architecture,
4. Research the marketplace for appropriate off-the-shelf systems and CASE tools to facilitate system delivery.

At PDR:

1. Provide the list of software products to be developed (see Preliminary Design Document),
2. Provide an overall schedule of major system milestones (see Section 6),
3. Provide a set of requirements which drive development planning decisions (see SOC System Requirements Document),

4. Provide a description of the processes which will bring about a successful software development effort (this document).

Pre-CDR:

1. Perform detailed system design,
2. Resolve design level TBDs.

At CDR:

1. Provide the detailed design of the SOC system,
2. Provide the schedule of delivery of specific system components.

5.1.2. Unit Test Planning

Unit testing is that level of test carried out by the developer when creating an individual module of code. It tests to insure that the module behaves deterministically and correctly for a given set of input conditions and exits gracefully in cases where errors occur. FAME will employ the use of built-in unit testing whereby every code module will have tests built in that allow future testers to perform some level of unit testing without the original developer being involved. Built-in unit testing also allows for streamlined regression testing if changes are made to related modules later on.

5.1.3. Integration Test Planning

Integration test will be performed on various levels of integration. As units are made available that address larger system functionality, they will be assembled and run through pre-determined test sequences to make sure they work as a unit.

An example of integration of sub-components is joining the data storage subsystem, the data retrieval layer and any of the components which require the retrieval of information. This integration level test allows the team to isolate larger and larger sections of the final product in advance of system test.

5.1.4. System Test Planning

System Test provides the opportunity to run an "end-to-end" test on the entire system using set data inputs and given a set of expected results. It tests the actual functionality of the system in an environment that is, ideally, an exact duplicate of the operational environment. The goal of System Test is to identify problem areas that only arise when the system is run end-to-end. These are often operational and load balancing issues, but it may show software flaws as well.

Depending on resources available at the time, as much as possible, the FAME SOC System Test will be conducted by personnel who have no vested interest in passing the test (i.e., not the FAME SOC software team). If sufficient resources are not available, the test will be conducted by the software team, but monitored by a representative of the user community acting as Quality Assurance Monitor. The software team must be available to the testers to support configuration, troubleshooting, fault assessment and suggestions for remediation.

5.1.5. Transition to Operational Environment

Once the software has passed System Test, it will be transitioned to the operational environment. This will be done by extracting the exact version of the source code for the software that had been tested, including make files, database definition scripts, data loading scripts, and any other files required to create the complete executable version of the system in the operational environment.

5.1.6. Management Review

The FAME Program Manager will review the test status at a (period TBD) Test Review Meeting. State of new, outstanding, and closed Software Problem Reports will be discussed.

5.2. Software Development Environment

5.2.1. Software Engineering Environment

FAME will use the Rose modeling and design product by Rational Software. Rose provides the tools and guidance needed to bring about a full object oriented design and implementation. It also provides round-trip engineering features – meaning it will create code from a design, as well as provide a design from code already written. This will help ensure that the developed code doesn't deviate from the initial design (by comparing as-built design with baseline design). Non object-oriented elements of the system will be incorporated into the design using OO "wrappers" around structured code.

5.2.2. Software Test Environment

Software Test Environment will be set up to be as similar to the production environment as possible. Similar hardware with identical capacities as well as identical operating systems is the ideal environment.

The use of a software test tool will be evaluated during the detailed design phase of the project.

5.2.3. Software Development Library

The software development library will contain all specification, design, and control documents required for development. These include the FAME SOC System Requirements, the MOC-SOC ICD, The Operational Concept Document, the Data Model, and any other documents used by the team. These documents will be kept in a 'read-only' state and will be changed only via the normal Configuration Management procedures.

5.2.4. Software Development Files

All source files, make files, data definition files, and any other files needed for development will be kept in a configuration controlled repository which will be maintained by the Configuration Manager or a designee.

5.3. *Requirements Analysis*

5.3.1. Analysis of User Input

The user community for the FAME SOC includes the scientists and engineers at the SOC facility at USNO. Input from all potential users of the system is valuable in determining the full set of specifications for the system. These users will be interviewed by the systems designers to determine the set of cases in which the system will operate. These "user stories" will be the fundamental material to be used in synthesizing the system requirements.

5.3.2. Operational Concept

The FAME SOC System will be operated at the SOC facility at USNO. The basic concept of the system is:

1. to accept the data stream from the ground station,
2. archive the data,
3. process the data (low-res) for instrument health and data quality indicators, and
4. process the data (high-res) for the final science data products.

The full description of the SOC Operational Concept is found in the SOC System Operations Concept Document.

5.3.3. System Requirements

The System Requirements are specified in the FAME SOC System Requirements Document.

5.4. *System Design Approach*

The system will be designed using object oriented methodology. Once high level science-case objects and processes have been identified, these will be translated into a system architecture design. The system architect will then allocate requirements to the Hardware, Software and Operations systems. Trade-offs will be made at this time to determine the best manner in which to satisfy system level requirements.

5.5. *Software Requirements Analysis*

Once the requirements that are specific to the software system are identified, they will be expanded to form the set of technical specifications for the design of the software system. Trade-offs will be revisited to ensure that the requirements are being addressed in the best and most efficient manner. Once the system engineer has determined the full set of requirements, it will be reviewed by the Configuration Control Board (CCB)(see Section 5.11.1). The CCB will advise the Project Manager as to the completeness and accuracy of the software requirements. The Project Manager must approve the software requirements specification before design may commence. The initial software requirements specification will be available for review at Preliminary Design Review (PDR), with revisions expected between PDR and CDR.

5.6. Software Design Approach

5.6.1. Design Decisions

Parts of the FAME SOC system will evolve over time. We anticipate that the order of processing as well as the demands on the data store will change, and we may not have all of the parameters in hand at the time of initial design. The CCB exists to review and advise on design decisions that might effect the overall system.

Decisions regarding the design of the system will be made according to the impact of the decision. It is the job of the System Engineer and the Software Development Manager to determine the proper level of decision-making. Decisions which directly effect the satisfaction of a system requirement must be approved by the Program Manager. Additionally, if the fulfillment of a particular requirement is causing an excessive amount of impact on schedule or cost, the necessity of that requirement must be discussed at the CCB for potential change or removal.

5.6.2. Detailed Design

There are many ways to deliver detailed design. In recent years, the industry trend has been toward an object oriented approach to design. The advantages of this method are the high level of modularity and maintainability of the code as well as an increased ability to isolate and fix code errors.

The detailed design will follow the object oriented methodology. Detailed design will include the object diagrams, state diagrams and event traces (how and when the various objects communicate and pass control) as needed to allow the developers to produce the final system. A schedule for completing detailed design is provided in Section 7.

5.7. Implementation and Unit Testing

5.7.1. Development

Development of the FAME SOC will be conducted at the USNO facility in Washington, DC, while development of the Simulator will be at the NOFS facility in Flagstaff, AZ. Development of the SOC will be in C++, while the Simulator will be TBD.

5.7.2. Unit Testing

Each production module will be tested by the engineer responsible for coding it. Unit testing will consist of:

1. Identification of input test data (both valid and invalid data) to test
    a. Satisfaction of requirements
    b. Correct functioning of the unit,
2. Identification of associated expected outputs,
3. Coding of built-in test procedures,
4. Running the test,
5. Evaluating results of valid data inputs,
6. Evaluating results of invalid data inputs (error checking and handling).

The Quality Assurance manager will verify unit testing. Unit testing will be considered successful once all test (including error) conditions are successfully met. Once a module has been successfully tested, it will be checked into the Configuration Management system and will not be changed without a change request approved by the Configuration Manager.

5.7.3. Revision and Retest

If revisions are required of the module, i.e. a change request has been approved by the Configuration Manager, the engineer will make the change and rerun the entire unit test suite before returning the module to the CM repository. Full compliance will be verified by the Quality Assurance Manager.

5.8. Integration Testing

5.8.1. Selection of Units for Integration

During design phase, the software team will identify subsystems that should be tested at a level in between unit test and system test. These groupings will be coded in a timeline such that they will be able to be integration tested while other coding activities continue. During the coding of a grouping, the system engineer will determine a set of test cases that are appropriate to test its functionality.

5.8.2. Integration Testing

Integration testing will follow the same basic process as Unit Testing, except that there are no built-in test routines for Integration Test:

1. Identification of input test data (both valid and invalid data) to test correct functioning of the grouping,
2. Identification of associated expected outputs,
3. Running the test,
4. Evaluating results of valid data inputs,
5. Evaluating results of invalid data inputs (error checking and handling).

The Quality Assurance manager will verify Integration Testing. Integration testing will be considered successful once all test (including error) conditions are successfully met.

5.8.3. Revision and Retest

If errors are found during Integration Test, the culpable module(s) will be recoded, Unit Tested and the Integration Test will be rerun.

5.8.4. Recording Integration Test Results

Once Integration Test is successful, the Quality Assurance Manager will note the successful completion in the QA log.

5.9. System Testing

5.9.1. Independence in System Test

It is highly desirable to have a completely independent entity conduct System Test. This insures that there is no bias when conducting the test and that someone not familiar with system operation can still make the system work. It also helps to insure that proper documentation has been supplied and that the system is ready for delivery.

System test on the FAME SOC will be conducted by TBD.

5.9.2. Test Environment

The test environment will be identical to the production environment. This environment is TBD, but will be identified during the system design phase and will be presented at CDR.

5.9.3. System Test Preparation

In preparation for System Test, the following procedure will be followed:

1. Preparation of the Test Environment (hardware setup, operating system setup and test, installation of ancillary software products and networking, etc.)
2. Acceptance of the Test Environment (checked by system engineer and quality assurance)
3. Extraction of the Test Code from the CM Repository
4. Extraction of Test Data from the CM Repository
5. Installation of Test Code and Data on the Test Environment.

At the end of this procedure, the system will be handed over to the System Test team for performance of the test.

5.9.4. System Test Performance

System Test will follow the same basic steps as for Integration Test, however System Test will consist of many separate test procedures, most of which will be derived from the Integration Test procedures.

1. Identification of input test data (both valid and invalid data) to test correct functioning of the system,
2. Identification of associated expected outputs,
3. Running the test,
4. Evaluating results of valid data inputs,
5. Evaluating results of invalid data inputs (error checking and handling).

System Test will identify two types of error:

1. Non-Critical Error – An error which will allow the continuation of a test procedure without recoding.
2. Critical Error – An error which necessitates immediate recoding of part of the code in order to continue any testing.

5.9.5. Revision and Retest

A particular procedure may encounter several Non-Critical Errors and all should be addressed before rerunning the test. A Critical Error will cause the test to halt and will require recoding and retesting before the System Test can resume.

It is important to note that recoding can take place while other System Tests are run, but System Test will only be considered completed when all tests are run end-to-end without error.

5.9.6. Recording System Test Results

The Quality Assurance Manager will record all completed tests in the QA log, as well as recording all errors.

5.10. Software Use Preparation

5.10.1.Version Description

There are two planned versions of the SOC software. The first is the Initial Operational Capability Version (v1.0) which will include the functionality to satisfy the Launch Critical Requirements described in Section 4.2.4 of this document. The second version is the Full Operational Capability (FOC) Version (v2.0) which will include the functionality to satisfy all requirements described in the FAME SOC System Requirements Document.

5.10.2.User Manuals

The FAME SOC development team will provide user manuals. These manuals will cover the operation of the FAME SOC system at a level which can be understood by someone with a general understanding of the operating systems in use and of the FAME mission. These user manuals will provide step-by-step operations procedures and recommendations for troubleshooting. They will be designed specifically for in-house use and not for general distribution.

5.10.3.Installation

The system will be installed at the USNO FAME SOC facility on the operational systems once it has been accepted by the project. The FAME SOC System development team will perform the installation and provide initial operations support.

5.11. Configuration Management

5.11.1.Configuration Identification

The FAME Configuration Manager (CM) will govern the configuration of the system. The CM will have the authority to allow changes to the design and to the code once it has been delivered. The CM will convene, at their discretion, the Configuration Control Board (CCB) which will advise the CM on configuration decisions.

5.11.2. Configuration Control

The development team will employ a professional configuration management system for maintaining controls on the design, documentation and code. Developers of these products must register all changes once the product has been placed under configuration control.

There are many systems available to help enforce configuration control. A candidate solution would be to use a commercial configuration management product such as CVS or PVCS to manage the actual tracking of changes and a problem reporting system like Cygnus' GNATS software to manage the trouble ticketing.

In order to reduce the chance that the CM process will hamper the efforts of the design and development effort, the Configuration Manager will review problem reports on an as-needed basis, convening the CCB at least once every week during production to review progress.

5.11.3. Configuration Auditing

The CM will periodically audit the configuration of the system by reviewing the change logs and design to insure the code is being developed to the specifications. The CM may delegate this responsibility. This audit will be to make certain that only those changes that are authorized by the CM are actually being made.

5.11.4. Version Control

Version Control will be monitored by the CM or their designee. There are currently two major versions of the FAME SOC system envisioned, IOC and FOC. There will be many sub-releases of the software for test purposes. These will be numbered as decimal versions of the major releases as follows:

Pre IOC release	v0.01 through v0.99
IOC release	v1.00
Post IOC, Pre FOC release	v1.01 through v1.99
FOC release	v2.00
Post FOC release	v2.01 and up

5.12. Quality Assurance

5.12.1. Independence in Quality Assurance

The Quality Assurance function will be satisfied in two ways.

1. Review of design and code in progress. This will consist of peer review (design/code walkthrough) and review of code for adherence to standards and application to requirements.
2. Independent Quality Assurance Audits. Once code is delivered into the test environment, and independent QA entity will check the code for standards compliance and satisfaction of CM requirements.

Independence of Quality Assurance will be satisfied by having the Independent Quality Assurance Audits done by someone who is not intimately involved in the development process, but familiar with it. In this way, they will not feel the need to bias their assessment of the compliance to QA standards.

5.12.2. Quality Assurance Evaluations

The following reviews will be performed at various stages in the design and development process:

1. Peer Design Reviews. In the peer design review, the designer will present their work to the other designers on the team. The design will be reviewed for:
    a. Completeness
    b. Adherence to Standards
    c. Satisfaction of Requirements
    d. Alternative Methods
2. Peer Code Walkthroughs. In these informal discussions, the coder will present their work to the rest of the team. The code will be reviewed for:
    a. Accurate Reflection of Design
    b. Adherence to Standards
    c. Alternative Methods
3. Independent Quality Assurance Audits. These audits, performed by an independent entity, will review:
    a. Adherence to Standards
    b. Compliance with CM Procedures
    c. Successful Completion of Unit Testing

5.13. Corrective Action

5.13.1. Software Problem Reports

SPRs may be generated by anyone involved in test or use of the system. SPRs will be maintained by a tracking system such as GNATS. SPRs will contain information on the nature of the problem, when the problem was encountered, who generated the SPR and recommendations for resolution of the problem.

5.13.2. Corrective Action Management

SPRs will be reviewed by the software manager. The software manager will assess the SPRs for difficulty, resolution action and priority. The CM will be the final authority on all corrective actions. Once the CM has approved a corrective action, the software manager will schedule time and resources commensurate with the difficulty and priority level assigned to the action.

5.14. Reviews

5.14.1. Technical Reviews

The FAME SOC system development team will conduct periodic technical reviews. These reviews will be open to all FAME personnel. At these reviews, the team will present the technical progress made since the previous review. Attendees will be invited to comment on progress and advise on plans. These technical reviews will be held TBD.

5.14.2.Management Reviews

The FAME SOC management team will conduct periodic management reviews. These reviews will be open to TBD. At these reviews, the management team will present the project status through schedules and delivery milestones. Additionally, schedule projections and upcoming events will be presented for review and comment. These management review will be held TBD.

5.15. Other Activities

5.15.1.Risk Management

There are three aspects of risk management – cost, schedule and technical. In software projects, cost risk is almost always driven by schedule performance and this appears to be the case with the FAME SOC. Schedule risk is typically driven by lack of understanding of the technical solution at design time. Therefore, it is our strategy to address overall risk by aggressively seeking to minimize the technical risk on the project.

The high degree of precision and accuracy desired from the FAME data requires the conceptualization and production of cutting edge algorithms for spatial positioning. In order to minimize the potential risk of this discovery of new ideas, the FAME team is currently engaged in prototype development of the most critical phases of the processing. These prototypes will be used as the basis for the production code once a clear process has been determined, tested and approved.

There are two types of technical risk:

1. Algorithm Uncertainty – we lack the understanding of exactly how the data will be processed in order to reach the science objectives. This is being addressed by prototyping of these processes, beginning with the highest risk ones. A schedule for this prototyping effort can be found in Appendix B of this document.
2. Technical Uncertainty – we will not understand the details of data storage and retrieval until the systems are in place and the actual functioning of the algorithms is known. This is being addressed by keeping the data storage separate from the actual algorithms thereby allowing the flexibility to "tweak" the storage solution as the system becomes more mature.

5.15.2.Security

The FAME SOC will be isolated from all network access other than the dedicated link to the MOC. This will all but eliminate the risk of communications security breaches. More detailed operational security measures will be determined as those requirements are identified.

5.15.3.Coordination with Associate Developers

TBD

6. Schedules

Appendix B shows the schedules for the prototype effort and for the detailed design effort.

7. Project Organization and Resources

7.1. Project Organization

7.2. Project Resources

7.2.1. Personnel

The FAME SOC software staff will be made up of the Software Manager and four developers. The exact skill mix of the developers will depend on decisions made during the design process. Some combination of algorithm expertise and C++ expertise will be required in order to successfully complete the project.

7.2.2. Facilities

The FAME SOC will be set up in Building 56 at the US Naval Observatory. Development of the SOC will take place at USNO/DC. Development of the simulator will be at USNO/Flagstaff.

7.2.3. Equipment

The FAME SOC will utilize equipment purchased for and dedicated entirely to the FAME SOC. The current proposed configuration is an array of Intel based multi-processor servers running LINUX connected via high-speed Ethernet with each other and with a RAID array of robust storage. UPSs will provide clean, consistent power and backups will be kept of all data stores on DVD media.

8. Notes

9. Appendix A.

Coding Standards for C++ Development

9.1. Introduction

These standards represent the formats to be used in coding C++ on the FAME SOC project. Code will be inspected as part of the QA process to ensure that these standards have been followed.

9.2. Variables

- All local variables will be lowercase and use uppercase letters to separate words within the name (e.g. myBlueHeaven).
- Class names will begin with an uppercase letter (e.g. BinaryStar).
- The wider the scope of the variable, the more descriptive it should be.
- Don’t use underscores in variable or function names.
- Don’t name variables with a leading underscore as this is used by systems to indicate non-standard extensions.
- Variables should not be reused for different purposes within the same routine.
- Avoid the use of pointers – they're confusing and often lead to unmaintainable code. Use references instead of pointers if possible. A reference cannot be changed to point to another object so that you know it always points to the same object.
- Avoid using the unsigned types if you don’t have to. The ‘unsigned’ keyword clutters function declarations and has no benefit if the extra headroom isn’t required.
- Never use global variables in projects larger than trivial. You can’t tell where or when they are used and they might have initialization sequence problems. Try using static class members and/or a singleton class to manage widely used variables.
- Constants should be defined in the smallest scope possible, for example if a constant is only used in one function it should be defined in the function.

9.3. Comments

- Clear code is preferable to well-commented messy code. Every effort should be made to make code clear through good design, simplicity, good naming and layout. Comments are definitely required where code is not clear.
- Avoid compulsory comments like author, date, class names, function names, and modification lists, as they get outdated. We will use a CM tool to insert these as needed.

9.4. Files

- Every class will have a separate source file and header file that matches the name of the class. For example class ProcessData should be defined in ProcessData.h and be located in ProcessData.cpp.
- The extension of the implementation file (eg .c .cc .cpp .cxx) will depend on your compiler (.cpp is the most likely).
- Use precompiled headers. This can greatly improve compilation speed, but also provides a consistent place to #include all system and library headers, so that they don't have to be included anywhere else.
- Avoid adding a full path or relative path to #include statements. Instead add the directory or root directory to the include directories list in the makefile. For example have "…/path/libraries/" as an include directory and have #include "LibrarySubDir/filename.h" in the precompiled header.

9.5. Clear Code Style

- There should be only one operation per line. For example the commonly seen line

```
if (doSomething()== false) ...
```

should be split into two lines.

```
result = doSomething();
if (result == false) ...
```

- Avoid using 'goto'.
- Functions must be explicitly typed (including `void`).
- Use explicit type casting.
- Do not use `char` variables in the place of `int` variables.
- Do not use operations in argument lists.
- Do not exceed 80 characters per line (use continuations)

9.6. Formatting & Layout

- Layout should be written for maximum code readability.
- Use #include <filename.h> for library & system headers.
- Use #include "filename.h" for non-system headers.
- Don't check equivalence backwards .
  e.g.: `if (false == variableA)`
  It may be safer but it is more difficult to read.
- Each variable declaration needs a line of its own.
- If the variable declaration is a pointer, place the star next to the type, not the name. Eg char* p not char *p;
- Header files should use the '#ifndef/#define/#endif' mechanism to prevent multiple inclusions. The defined name should be of the form _Filename_H.
- Macros (if you use them) should be in upper case.

- Use spaces instead of tabs in a source file.
- Functions and variables should be mixed case and begin with a lower case letter.
- Place a space between operators such as + and ||.
- Use parentheses wherever they may remove possible ambiguity in the minds of the reader.
- Matching braces (‘{‘ ’}’) should line up vertically and be on their own horizontal line.
- Place single spaces immediately after commas in routine parameter lists.

9.7. Safe C++ coding

- Routines and their parameters should be declared 'const' wherever possible.
- Use the ‘assert’ macro to test assumed or expected conditions.
- Classes that will be inherited from should have a virtual destructor.
- If a class defines either an assignment operator or a copy constructor it should define both.
- Don’t throw objects allocated from the heap. You don’t know if they will be freed.
- If you catch an exception, make sure you have a default `catch(...)`so that all exceptions are caught at that point.
- If you are writing a library or subsystem that will be used by a different project, it must be in a namespace.
- Use set/get functions instead of public variables. If you have a `getCount()` function instead of a public member you can add a breakpoint to the function to see when it is being used. If you have a lot of public variables, consider a structure without code and separate the code from the data.

9.8. Reducing C++ complexity

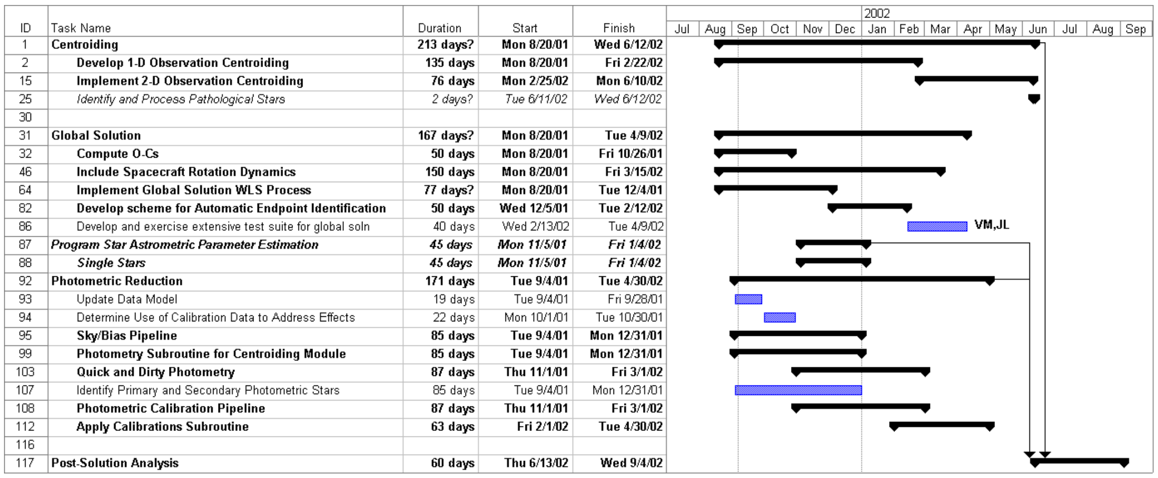
- Poorly written C++ can be very difficult to understand. It can be simplified by not using many of the features of C++: some of which are leftover from C, some which are poorly understood by many developers and some which are unnecessary. Keep it simple. Elegant design is always better than complex code.
- #ifdef/#endif should not be used in a function to make it portable. Use wrapper classes to cover platform-dependent code.
- Avoid macros. Always use constants or functions if possible as they are clearer and are more easily understood by software tools such as debuggers and source browsers.
- Avoid c library functions which are superceded by modern objects, e.g. use string class methods rather than strcmp, use streams instead of c runtime functions.
- Always try to use STL classes over proprietary or third party classes.
- Use if statements instead of ?:.
- Avoid excessive use of templates. Templates are excellent for utility classes such as collections but abstract base classes are usually a better design and are clearer to understand.
- Avoid operator overloading. Using a method on an object is much clearer and easier to follow than a called operator.

9.9. Portability

- Try to avoid platform specific API calls. This will depend on the type of application, but a generic program core should call objects that can polymorphically adapt to different platforms.
- Avoid pointer arithmetic.
- Never assume a pointer length is a given length, and then don't use it anyway.
- Use long instead of int.

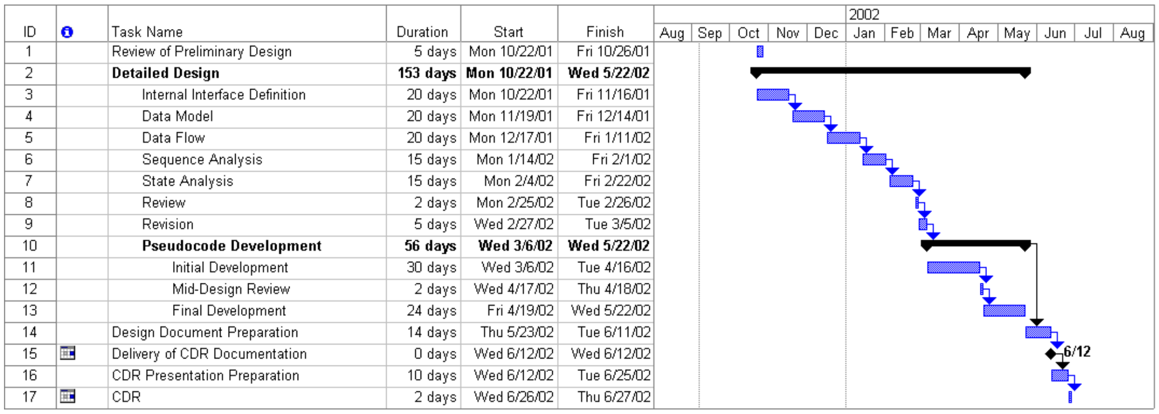
10. Appendix B

10.1. Pipeline Prototype Schedule



ID	Task Name	Duration	Start	Finish
1	**Centroiding**	**213 days?**	**Mon 8/20/01**	**Wed 6/12/02**
2	**Develop 1-D Observation Centroiding**	**135 days**	**Mon 8/20/01**	**Fri 2/22/02**
15	**Implement 2-D Observation Centroiding**	**76 days**	**Mon 2/25/02**	**Mon 6/10/02**
25	*Identify and Process Pathological Stars*	*2 days?*	*Tue 6/11/02*	*Wed 6/12/02*
30				
31	**Global Solution**	**167 days?**	**Mon 8/20/01**	**Tue 4/9/02**
32	**Compute O-Cs**	**50 days**	**Mon 8/20/01**	**Fri 10/26/01**
46	**Include Spacecraft Rotation Dynamics**	**150 days**	**Mon 8/20/01**	**Fri 3/15/02**
64	**Implement Global Solution WLS Process**	**77 days?**	**Mon 8/20/01**	**Tue 12/4/01**
82	**Develop scheme for Automatic Endpoint Identification**	**50 days**	**Wed 12/5/01**	**Tue 2/12/02**
86	Develop and exercise extensive test suite for global soln	40 days	Wed 2/13/02	Tue 4/9/02
87	***Program Star Astrometric Parameter Estimation***	***45 days***	***Mon 11/5/01***	***Fri 1/4/02***
88	***Single Stars***	***45 days***	***Mon 11/5/01***	***Fri 1/4/02***
92	**Photometric Reduction**	**171 days**	**Tue 9/4/01**	**Tue 4/30/02**
93	Update Data Model	19 days	Tue 9/4/01	Fri 9/28/01
94	Determine Use of Calibration Data to Address Effects	22 days	Mon 10/1/01	Tue 10/30/01
95	**Sky/Bias Pipeline**	**85 days**	**Tue 9/4/01**	**Mon 12/31/01**
99	**Photometry Subroutine for Centroiding Module**	**85 days**	**Tue 9/4/01**	**Mon 12/31/01**
103	**Quick and Dirty Photometry**	**87 days**	**Thu 11/1/01**	**Fri 3/1/02**
107	Identify Primary and Secondary Photometric Stars	85 days	Tue 9/4/01	Mon 12/31/01
108	**Photometric Calibration Pipeline**	**87 days**	**Thu 11/1/01**	**Fri 3/1/02**
112	**Apply Calibrations Subroutine**	**63 days**	**Fri 2/1/02**	**Tue 4/30/02**
116				
117	**Post-Solution Analysis**	**60 days**	**Thu 6/13/02**	**Wed 9/4/02**

10.2. Design Schedule



ID	ℹ	Task Name	Duration	Start	Finish
1		Review of Preliminary Design	5 days	Mon 10/22/01	Fri 10/26/01
2		**Detailed Design**	**153 days**	**Mon 10/22/01**	**Wed 5/22/02**
3		Internal Interface Definition	20 days	Mon 10/22/01	Fri 11/16/01
4		Data Model	20 days	Mon 11/19/01	Fri 12/14/01
5		Data Flow	20 days	Mon 12/17/01	Fri 1/11/02
6		Sequence Analysis	15 days	Mon 1/14/02	Fri 2/1/02
7		State Analysis	15 days	Mon 2/4/02	Fri 2/22/02
8		Review	2 days	Mon 2/25/02	Tue 2/26/02
9		Revision	5 days	Wed 2/27/02	Tue 3/5/02
10		**Pseudocode Development**	**56 days**	**Wed 3/6/02**	**Wed 5/22/02**
11		Initial Development	30 days	Wed 3/6/02	Tue 4/16/02
12		Mid-Design Review	2 days	Wed 4/17/02	Thu 4/18/02
13		Final Development	24 days	Fri 4/19/02	Wed 5/22/02
14		Design Document Preparation	14 days	Thu 5/23/02	Tue 6/11/02
15	▦	Delivery of CDR Documentation	0 days	Wed 6/12/02	Wed 6/12/02
16		CDR Presentation Preparation	10 days	Wed 6/12/02	Tue 6/25/02
17	▦	CDR	2 days	Wed 6/26/02	Thu 6/27/02